05 | Tensor变形记：快速掌握Tensor切分、变形等方法

2021-10-20 方远

《PyTorch深度学习实战》 课程介绍 >

讲述：方远
时长 15:38 大小 14.32M

你好，我是方远。

上节课我们一起学习了 Tensor 的基础概念，也熟悉了创建、转换、维度变换等操作，掌握了这些基础知识，你就可以做一些简单的 Tensor 相关的操作了。

不过，要想在实际的应用中更灵活地用好 Tensor，Tensor 的连接、切分等操作也是必不可少的。今天这节课，咱们就通过一些例子和图片来一块学习下。虽然这几个操作比较有难度，但只要你耐心听我讲解，然后上手练习，还是可以拿下的。

Tensor 的连接操作

在项目开发中，深度学习某一层神经元的数据可能有多个不同的来源，那么就需要将数据进行组合，这个组合的操作，我们称之为**连接**。

cat

连接的操作函数如下。

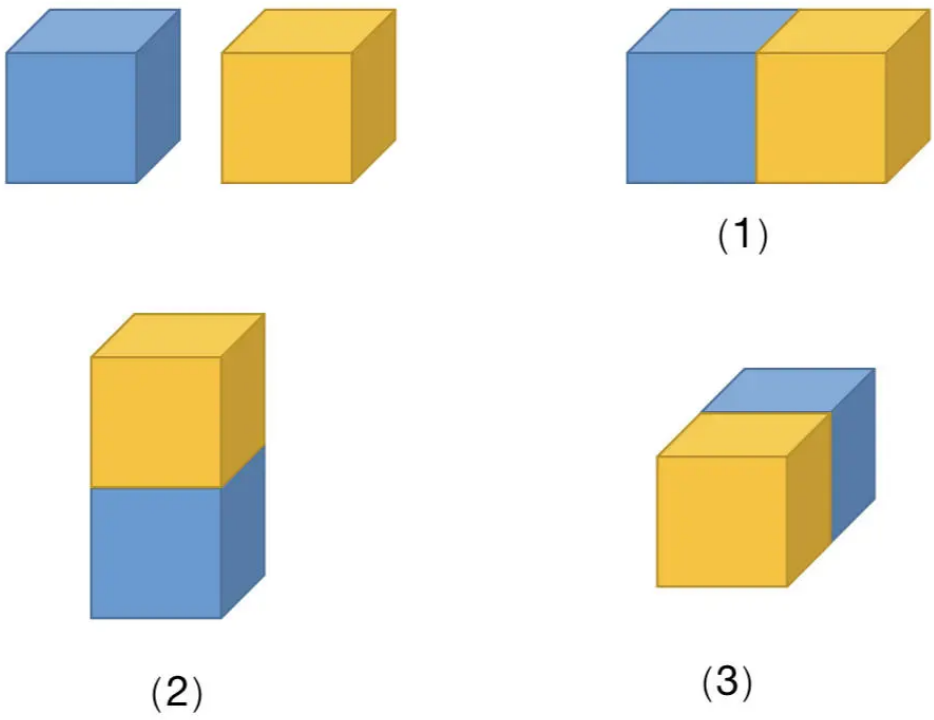
```
1 torch.cat(tensors, dim = 0, out = None)
```

复制代码

cat 是 concatnate 的意思，也就是拼接、联系的意思。该函数有两个重要的参数需要你掌握。

第一个参数是 tensors，它很好理解，就是若干个我们准备进行拼接的 Tensor。

第二个参数是 dim，我们回忆一下 Tensor 的定义，Tensor 的维度（秩）是有多种情况的。比如有两个 3 维的 Tensor，可以有几种不同的拼接方式（如下图），dim 参数就可以对此作出约定。



极客时间

看到这里，你可能觉得上面画的图是三维的，看起来比较晦涩，所以咱们先从简单的二维的情况说起，我们先声明两个 3x3 的矩阵，代码如下：

复制代码

```
>>> A=torch.ones(3,3)
>>> B=2*torch.ones(3,3)
>>> A
tensor([[1., 1., 1.],
        [1., 1., 1.],
        [1., 1., 1.]])
>>> B
tensor([[2., 2., 2.],
        [2., 2., 2.],
        [2., 2., 2.]])
```

我们先看看 dim=0 的情况，拼接的结果是怎样的：

复制代码

```
>>> C=torch.cat((A,B),0)
>>> C
tensor([[1., 1., 1.],
        [1., 1., 1.],
        [1., 1., 1.],
        [2., 2., 2.],
        [2., 2., 2.],
        [2., 2., 2.]])

```

你会发现，两个矩阵是按照“行”的方向拼接的。

我们接下来再看看，dim=1 的情况是怎样的：

复制代码

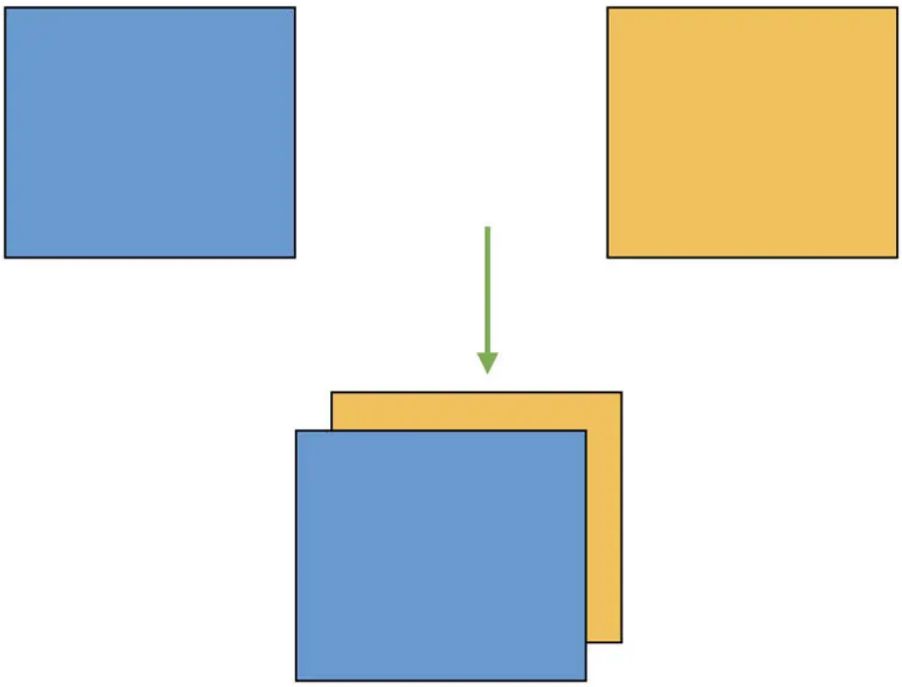
```
>>> D=torch.cat((A,B),1)
>>> D
tensor([[1., 1., 1., 2., 2., 2.],
        [1., 1., 1., 2., 2., 2.],
        [1., 1., 1., 2., 2., 2.]])
```

显然，两个矩阵，是按照“列”的方向拼接的。那如果 Tensor 是三维甚至更高维度的呢？其实道理也是一样的，dim 的数值是多少，两个矩阵就会按照相应维度的方向链接两个 Tensor。

看到这里你可能会问了，cat 实际上是将多个 Tensor 在已有的维度上进行连接，那如果想增加新的维度进行连接，又该怎么做呢？这时候就需要 stack 函数登场了。

stack

为了让你加深理解，我们还是结合具体例子来看看。假设我们有两个二维矩阵 Tensor，把它们“堆叠”放在一起，构成一个三维的 Tensor，如下图：



这相当于原来的维度（秩）是 2，现在变成了 3，变成了一个立体的结构，增加了一个维度。你需要注意的是，这跟前面的 cat 不同，cat 中示意图的例子，原来就是 3 维的，cat 之后仍旧是 3 维的，而现在咱们是**从 2 维变成了 3 维**。

在实际图像算法开发中，咱们有时候需要将多个单通道 Tensor（2 维）合并，得到多通道的结果（3 维）。而实现这种增加维度拼接的方法，我们把它叫做 stack。

stack 函数的定义如下：

复制代码

```
torch.stack(inputs, dim=0)
```

其中，inputs 表示需要拼接的 Tensor，dim 表示新建立维度的方向。

那 stack 如何使用呢？我们一块来看一个例子：

复制代码

```
>>> A=torch.arange(0,4)
>>> A
tensor([0, 1, 2, 3])
>>> B=torch.arange(5,9)
>>> B
tensor([5, 6, 7, 8])
>>> C=torch.stack((A,B),0)
>>> C
tensor([[0, 1, 2, 3],
        [5, 6, 7, 8]])
>>> D=torch.stack((A,B),1)
>>> D
tensor([[0, 5],
        [1, 6],
        [2, 7],
        [3, 8]])
```

结合代码，我们可以看到，首先我们构建了两个 4 元素向量 A 和 B，它们的维度是 1。然后，我们在 dim=0，也就是“行”的方向上新建一个维度，这样维度就成了 2，也就得到了 C。而对于 D，我们则是在 dim=1，也就是“列”的方向上新建维度。

Tensor 的切分操作

学完了连接操作之后，我们再来看看连接的逆操作：**切分**。

切分就是连接的逆过程，有了刚才的经验，你很容易就会想到，切分的操作也应该有很多种，比如切片、切块等。没错，切分的操作主要分为三种类型：chunk、split、unbind。

乍一看有不少，其实是因为它们各有特点，适用于不同的使用情景，让我们一起看一下。

chunk

chunk 的作用就是将 Tensor 按照声明的 dim，进行尽可能平均的划分。

比如说，我们有一个 32channel 的特征，需要将其按照 channel 均匀分成 4 组，每组 8 个 channel，这个切分就可以通过 chunk 函数来实现。具体函数如下：

复制代码

```
torch.chunk(input, chunks, dim=0)
```

我们挨个来看看函数中涉及到的三个参数：

首先是 input，它表示要做 chunk 操作的 Tensor。

接着，我们看下 chunks，它代表将要被划分的块的数量，而不是每组的数量。请注意，**chunks 必须是整型**。

最后是 dim，想想这个参数是什么意思呢？对，就是按照哪个维度来进行 chunk。

还是跟之前一样，我们通过几个代码例子直观感受一下。我们从一个简单的一维向量开始：

复制代码

```
>>> A=torch.tensor([1,2,3,4,5,6,7,8,9,10])
>>> B = torch.chunk(A, 2, 0)
>>> B
(tensor([1, 2, 3, 4, 5]), tensor([ 6,  7,  8,  9, 10]))
```

这里我们通过 chunk 函数，将原来 10 位长度的 Tensor A，切分成了两个一样 5 位长度的向量。（注意，B 是两个切分结果组成的 tuple）。

那如果 chunk 参数不能够整除的话，结果会是怎样的呢？我们接着往下看：

复制代码

```
>>> B = torch.chunk(A, 3, 0)
```

```
>>> B
(tensor([1. 2. 3. 4]). tensor([5. 6. 7. 8]). tensor([ 9. 10]))
```

我们发现，10 位长度的 Tensor A，切分成了三个向量，长度分别是 4，4，2 位。这是怎么分的呢，不应该是 3，3，4 这样更为平均的方式么？

想要解决问题，就得找到规律。让我们再来看一个更大一点的例子，将 A 改为 17 位长度。

复制代码

```
>>> A=torch.tensor([1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17])
>>> B = torch.chunk(A, 4, 0)
>>> B
(tensor([1, 2, 3, 4, 5]), tensor([ 6,  7,  8,  9, 10]), tensor([11, 12, 13, 14
```

17 位长度的 Tensor A，切分成了四个分别为 5，5，5，2 位长度的向量。这时候你就会发现，其实在计算每个结果元素个数的时候，chunk 函数是先做除法，然后再向上取整得到每组的数量。

比如上面这个例子，17/4=4.25，向上取整就是 5，那就先逐个生成若干个长度为 5 的向量，最后不够的就放在一块，作为最后一个向量（长度 2）。

那如果 chunk 参数大于 Tensor 可以切分的长度，又要怎么办呢？我们实际操作一下，代码如下：

复制代码

```
>>> A=torch.tensor([1,2,3])
>>> B = torch.chunk(A, 5, 0)
>>> B
(tensor([1]), tensor([2]), tensor([3]))
```

显然，被切分的 Tensor 只能分成若干个长度为 1 的向量。

由此可以推论出二维的情况，我们再举一个例子， 看看二维矩阵 Tensor 的情况 ：

```
>>> A=torch.ones(4,4)
>>> A
tensor([[1., 1., 1., 1.],
        [1., 1., 1., 1.],
        [1., 1., 1., 1.],
        [1., 1., 1., 1.]])
>>> B = torch.chunk(A, 2, 0)
>>> B
(tensor([[1., 1., 1., 1.],
        [1., 1., 1., 1.]]),
tensor([[1., 1., 1., 1.],
        [1., 1., 1., 1.]]))
```

复制代码

还是跟前面的 cat 一样，这里的 dim 参数，表示的是第 dim 维度方向上进行切分。

刚才介绍的 chunk 函数，是按照“切分成确定的份数”来进行切分的，那如果想按照“每份按照确定的大小”来进行切分，该怎样做呢？PyTorch 也提供了相应的方法，叫做 split。

split

split 的函数定义如下，跟前面一样，我们还是分别看看这里涉及的参数。

```
torch.split(tensor, split_size_or_sections, dim=0)
```

复制代码

首先是 tensor，也就是待切分的 Tensor。

然后是 split_size_or_sections 这个参数。当它为整数时，表示将 tensor 按照每块大小为这个整数的数值来切割；当这个参数为列表时，则表示将此 tensor 切成和列表中元素一样大小的块。

最后同样是 dim，它定义了要按哪个维度切分。

同样的，我们举几个例子来看一下 split 的具体操作。首先是 split_size_or_sections 是整数的情况。

```
>>> A=torch.rand(4,4)
>>> A
tensor([[0.6418, 0.4171, 0.7372, 0.0733],
        [0.0935, 0.2372, 0.6912, 0.8677],
        [0.5263, 0.4145, 0.9292, 0.5671],
        [0.2284, 0.6938, 0.0956, 0.3823]])
>>> B=torch.split(A, 2, 0)
>>> B
(tensor([[0.6418, 0.4171, 0.7372, 0.0733],
        [0.0935, 0.2372, 0.6912, 0.8677]]),
tensor([[0.5263, 0.4145, 0.9292, 0.5671],
        [0.2284, 0.6938, 0.0956, 0.3823]]))
```

复制代码

在这个例子里，我们看到，原来 4x4 大小的 Tensor A，沿着第 0 维度，也就是沿“行”的方向，按照每组 2“行”的大小进行切分，得到了两个 2x4 大小的 Tensor。

那么问题来了，如果 split_size_or_sections 不能整除对应方向的大小的话，会有怎样的结果呢？我们将代码稍作修改就好了：

```
>>> C=torch.split(A, 3, 0)
>>> C
(tensor([[0.6418, 0.4171, 0.7372, 0.0733],
        [0.0935, 0.2372, 0.6912, 0.8677],
        [0.5263, 0.4145, 0.9292, 0.5671]]),
tensor([[0.2284, 0.6938, 0.0956, 0.3823]]))
```

复制代码

根据刚才的代码我们就能发现，原来，PyTorch 会尽可能凑够每一个结果，使得其对应 dim 的数据大小等于 split_size_or_sections。如果最后剩下的不够，那就把剩下的内容放到一块，作为最后一个结果。

接下来，我们再看一下 split_size_or_sections 是列表时的情况。刚才提到了，当 split_size_or_sections 为列表的时候，表示将此 tensor 切成和列表中元素大小一样的大小的块，我们来看一段对应的代码：

```
>>> A=torch.rand(5,4)
>>> A
tensor([[0.1005, 0.9666, 0.5322, 0.6775],
        [0.4990, 0.8725, 0.5627, 0.8360],
```

复制代码

```
        [0.3427, 0.9351, 0.7291, 0.7306],
        [0.7939, 0.3007, 0.7258, 0.9482],
        [0.7249, 0.7534, 0.0027, 0.7793]])
>>> B=torch.split(A,(2,3),0)
>>> B
(tensor([[0.1005, 0.9666, 0.5322, 0.6775],
        [0.4990, 0.8725, 0.5627, 0.8360]]),
tensor([[0.3427, 0.9351, 0.7291, 0.7306],
        [0.7939, 0.3007, 0.7258, 0.9482],
        [0.7249, 0.7534, 0.0027, 0.7793]]))
```

这部分代码怎么解释呢？其实也很好理解，就是将 Tensor A，沿着第 0 维进行切分，每一个结果对应维度上的尺寸或者说大小，分别是 2（行），3（行）。

unbind

通过学习前面的几个函数，咱们知道了怎么按固定大小做切分，或者按照索引 index 来进行选择。现在我们想象一个应用场景，如果我们现在有一个 3 channel 图像的 Tensor，想要逐个获取每个 channel 的数据，该怎么做呢？

假如用 chunk 的话，我们需要将 chunks 设为 3；如果用 split 的话，需要将 split_size_or_sections 设为 1。

虽然它们都可以实现相同的目的，但是如果 channel 数量很大，逐个去取也比较折腾。这时候，就需要用到另一个函数：unbind，它的函数定义如下：

复制代码

```
torch.unbind(input, dim=0)
```

其中，input 表示待处理的 Tensor，dim 还是跟前面的函数一样，表示切片的方向。

我们结合例子来理解：

复制代码

```
>>> A=torch.arange(0,16).view(4,4)
>>> A
tensor([[ 0,  1,  2,  3],
        [ 4,  5,  6,  7],
        [ 8,  9, 10, 11],
```

```
        [12, 13, 14, 15]])
>>> b=torch.unbind(A, 0)
>>> b
(tensor([0, 1, 2, 3]),
tensor([4, 5, 6, 7]),
tensor([ 8,  9, 10, 11]),
tensor([12, 13, 14, 15]))
```

在这个例子中，我们首先创建了一个 4x4 的二维矩阵 Tensor，随后我们从第 0 维，也就是“行”的方向进行切分 ，因为矩阵有 4 行，所以就会得到 4 个结果。

接下来，我们看一下：如果从第 1 维，也就是“列”的方向进行切分，会是怎样的结果呢：

复制代码

```
>>> b=torch.unbind(A, 1)
>>> b
(tensor([ 0,  4,  8, 12]),
tensor([ 1,  5,  9, 13]),
tensor([ 2,  6, 10, 14]),
tensor([ 3,  7, 11, 15]))
```

不难发现，这里是按照“列”的方向进行拆解的。所以，**unbind 是一种降维切分的方式**，相当于删除一个维度之后的结果。

Tensor 的索引操作

你有没有发现，刚才我们讲的 chunk 和 split 操作，我们都是将数据整体进行切分，并获得全部结果。但有的时候，我们只需要其中的一部分，这要怎么做呢？一个很自然的想法就是，直接告诉 Tensor 我想要哪些部分，这种方法我们称为索引操作。

索引操作有很多方式，有提供好现成 API 的，也有用户自行定制的操作，其中最常用的两个操作就是 index_select 和 masked_select，我们分别去看看用法。

index_select

这里就需要 index_select 这个函数了，其定义如下：

```
torch.index_select(tensor, dim, index)
```

复制代码

这里的 tensor、dim 跟前面函数里的一样，不再赘述。我们重点看一看 index，它表示从 dim 维度中的哪些位置选择数据，这里需要注意，index**是 torch.Tensor 类型**。

还是跟之前一样，我们来看几个示例代码：

复制代码

```
>>> A=torch.arange(0,16).view(4,4)
>>> A
tensor([[ 0,  1,  2,  3],
        [ 4,  5,  6,  7],
        [ 8,  9, 10, 11],
        [12, 13, 14, 15]])
>>> B=torch.index_select(A,0,torch.tensor([1,3]))
>>> B
tensor([[ 4,  5,  6,  7],
        [12, 13, 14, 15]])
>>> C=torch.index_select(A,1,torch.tensor([0,3]))
>>> C
tensor([[ 0,  3],
        [ 4,  7],
        [ 8, 11],
        [12, 15]])
```

在这个例子中，我们先创建了一个 4x4 大小的矩阵 Tensor A。然后，我们从第 0 维选择第 1（行）和 3（行）的数据，并得到了最终的 Tensor B，其大小为 2x4。随后我们从 Tensor A 中选择第 0（列）和 3（列）的数据，得到了最终的 Tensor C，其大小为 4x2。

怎么样，是不是非常简单？

masked_select

刚才介绍的 indexed_select，它是基于给定的索引来进行数据提取的。但有的时候，我们还想通过一些判断条件来进行选择，比如提取深度学习网络中某一层中数值大于 0 的参数。

这时候，就需要用到 PyTorch 提供的 masked_select 函数了，我们先来看它的定义：

复制代码

```
torch.masked_select(input, mask, out=None)
```

这里我们只需要关心前两个参数，input 和 mask。

input 表示待处理的 Tensor。mask 代表掩码张量，也就是满足条件的特征掩码。这里你需要注意的是，mask 须跟 input 张量有相同数量的元素数目，但形状或维度不需要相同。

你是不是还感觉有些云里雾里？让我来举一个例子，你看了之后，一下子就能明白。

你在平时的练习中有没有想过，如果我们让 Tensor 和数字做比较，会有什么样的结果？比如后面这段代码，我们随机生成一个 5 位长度的 Tensor A：

复制代码

```
>>> A=torch.rand(5)
>>> A
tensor([0.3731, 0.4826, 0.3579, 0.4215, 0.2285])
>>> B=A>0.3
>>> B
tensor([ True,  True,  True,  True, False])
```

在这段代码里，我们让 A 跟 0.3 做比较，得到了一个新的 Tensor，内部每一个数值表示的是 A 中对应数值是否大于 0.3。

比如第一个数值原来是 0.3731，大于 0.3，所以是 True；最后一个数值 0.2285 小于 0.3，所以是 False。

这个新的 Tensor 其实就是一个掩码张量，它的每一位表示了一个判断条件是否成立的结果。

然后，我们继续写一段代码，看看基于掩码 B 的选择是怎样的结果 ：

复制代码

```
>>> C=torch.masked_select(A, B)
>>> C
```

```
3 tensor([0.3731, 0.4826, 0.3579, 0.4215])
```

你会发现，C 实际上得到的就是：A 中“**满足 B 里面元素值为 True 的**”对应位置的数据。

好了，这下你应该知道了 masked_select 的作用了吧？其实就是我们根据要筛选的条件，得到一个掩码张量，然后用这个张量去提取 Tensor 中的数据。

根据这个思路，上面的例子就可以简化为：

复制代码

```
1 >>> A=torch.rand(5)
2 >>> A
3 tensor([0.3731, 0.4826, 0.3579, 0.4215, 0.2285])
4 >>> C=torch.masked_select(A, A>0.3)
5 >>> C
6 tensor([0.3731, 0.4826, 0.3579, 0.4215])
```

是不是非常简单呢？

小结

恭喜你完成了这节课的学习。这节课，我们一同学习了 Tensor 里更加高级的操作，包括 Tensor 之间的连接操作，Tensor 内部的切分操作，以及基于索引或者筛选条件的数据选择操作。

当然了，在使用这些函数的时候，你最需要关注的就是边界的数值大小，具体来说就是维度和大小相关的参数，一定要提前仔细计算好，要不然就会产生错误的结果。

结合众多的例子，我相信你一定可以拿下这些操作。

这里我特意给你梳理了一张表格，总结归纳了 Tensor 中的主要函数跟用法。不过这些参数咱们也不用死记硬背，我们在使用的时候，根据需要灵活查询相关的参数列表即可。

函数名	功能	使用要点/适用的情况
torch.tensor	直接创建Tensor	一般用于简单Tensor的创建
torch.from_numpy	从NumPy中创建	基于已有的NumPy数据进行Tensor创建
torch.zeros	创建零矩阵Tensor	特殊形状或者数值分布的Tensor的创建
torch.ones	创建全一矩阵Tensor	
torch.rand	创建随机分布Tensor	
torch.randn		
torch.normal		
torch.randint		
tensor.shape	获取Tensor形状	注意区分函数和属性的区别
tensor.size		
tensor.permute	Tensor转秩	transpose每次只能转换两个维度
tensor.transpose	Tensor维度交换	
tensor.view	Tensor 变形	view不能处理内存不连续Tensor的结构
tensor.reshape	Tensor变形	reshape可以处理内存不连续Tensor的结构
tensor.squeeze	Tensor删除维度	———
tensor.unsqueeze	Tensor增加维度	———
torch.cat	Tensor拼接	不增加维度
torch.stack	Tensor堆叠	增加维度
torch.chunk	Tensor尽可能平均切分	———
torch.split	Tensor按固定大小切分	———
torch.unbind	Tensor按维度切分	———
torch.index_select	Tensor 选择	基于给定的索引来进行数据提取
torch.masked_select		通过一些判断条件来进行选择

极客时间

通过这两节课，我们搞懂了 Tensor 的一系列操作，在以后的项目中，你就可以游刃有余地对 Tensor 进行各种花式操作了，加油!

每课一练

现在有个 Tensor，如下：

复制代码

```
>>> A=torch.tensor([[4,5,7], [3,9,8],[2,3,4]])
>>> A
tensor([[4, 5, 7],
        [3, 9, 8],
        [2, 3, 4]])
```

我们想提取出其中第一行的第一个，第二行的第一、第二个，第三行的最后一个，该怎么做呢？

欢迎你在留言区跟我交流互动，也推荐你把这节课分享给更多同事、朋友！

分享给需要的人，Ta订阅后你可得 20 元现金奖励

生成海报并分享

赞 2 提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。 页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 04 | Tensor：PyTorch中最基础的计算单元

下一篇 06 | Torchvision（上）：数据读取，训练开始的第一步

精选留言 (7)

写留言

lwg0452
2021-10-20

更正😬
mask = torch.tensor([[1, 0, 0], [1, 1, 0], [0, 0, 1]])

B = torch.masked_select(A, mask>0)

展开

作者回复: 你好，lwg0452。👍👍👍👍👍👍回答正确^^。

1

王骥

2021-10-24

input 表示待处理的 Tensor。mask 代表掩码张量，也就是满足条件的特征掩码。这里你需要注意的是，mask 须跟 input 张量有相同数量的元素数目，但形状或维度不需要相同。

老师，mask 须跟 input 张量有相同数量的元素数目，但形状或维度不需要相同 这句话该怎么理解？...

展开

作者回复: 你好，王骥。感谢你的留言。很赞的问题，这里“形状或维度不需要相同”，我们举个二维矩阵的例子，你可以只写“一行”，或者所有“行”，但是“列”的数量必须跟原来的数量一致。如下代码：

```
a=torch.randn(3,5)
#不会报错
mask=torch.tensor([True, False,False,True,True])
torch.masked_select(a,mask)

#会报错
mask=torch.tensor([True, False,False,True])
torch.masked_select(a,mask)
```

此外，需要注意的是，官网强调：The shapes of the mask tensor and the input tensor don’t need to match, but they must be broadcastable。不过一般而言咱们用的变量基本多为broadcastable的。

1

李雄

2021-10-23

喜欢这节的内容。

展开

作者回复: 你好，李雄。感谢你的留言，也谢谢你的认可。希望能与你一起学习进步^^

1

vcjmhg

2021-10-21

A = torch.tensor([[4, 5, 7], [3, 9, 8], [2, 3, 4]])
mask_matrix = torch.tensor([[1, 0, 0], [1, 1, 0], [0, 0, 1]])
B = torch.masked_select(A, mask == 1)

作者回复: 你好，vcjmhg。👍👍👍，每次的回答都很赞^^，加油~
不过masked_select中的参数是mask_matrix不是mask。：）

栗白

2021-10-20

A=torch.tensor([[4,5,7], [3,9,8],[2,3,4]])
B=torch.tensor([[1,0,0],[1,1,0],[0,0,1]])
C=torch.masked_select(A,B>0)

展开

作者回复: 你好，栗白。谢谢留言。👍👍👍👍👍👍👍👍完美。^^

narsil的梦

2021-10-20

练习：
A = torch.tensor([[4, 5, 7], [3, 9, 8], [2, 3, 4]])

# 提取出其中第一行的第一个，第二行的第一、第二个，第三行的最后一个
B = torch.chunk(A, 3, dim=0)...

展开

作者回复: 你好，narsil的梦，感谢你的留言。tensor的格式呀，咱们现在做的就是tensor的操作

1

lwg0452

2021-10-20

思考题 B = torch.masked_select(A, torch.eye(3) > 0)